

Fairly standard ones

$x \& (x-1)$
 = x with the lowest set bit cleared.
 $x \& \sim(x-1)$
 = extracts the lowest set bit of x (all others are clear). Pretty patterns when applied to a linear sequence.
 $x \& (x+(1<<n))$
 = x, with the run of set bits (possibly length 0) starting at bit n cleared.
 $x \& \sim(x+(1<<n))$
 = the run of set bits (possibly length 0) in x, starting at bit n.
 $x | (x+1)$
 = x with the lowest cleared bit set.
 $x | \sim(x+1)$
 = extracts the lowest cleared bit of x (all others are set).
 $x | (x-(1<<n))$
 = x, with the run of cleared bits (possibly length 0) starting at bit n set.
 $x | \sim(x-(1<<n))$
 = the lowest run of cleared bits (possibly length 0) in x, starting at bit n are the only clear bits.

Gray code

$x \wedge (x >> 1)$
 = The standard (binary-reflected) Gray code for x.
 $z = y = x; \text{ do } \{ y = y >> 1; z \wedge= y \} \text{ while } (y); \text{ return } z;$
 = decoder for the above Gray code.

NUL finding

from "newlib": Note the odd use of NULL for the NUL character.

```

#if LONG_MAX == 2147483647L
#define DETECTNULL(X) (((X) - 0x01010101) & ~(X) & 0x80808080)
#else
#if LONG_MAX == 9223372036854775807L
/* Nonzero if X (a long int) contains a NULL byte. */
#define DETECTNULL(X) (((X) - 0x0101010101010101) & ~(X) & 0x8080808080808080)
#else
#error long int is not a 32bit or 64bit type.
#endif
#endif

```

Take the case of the one byte equivalent:

$(X - 0x01) \& \sim(X) \& 0x80$
 $== \sim(x | \sim(x - 1)) \& 0x80$

From above we can see that the the first part of the expression is the lowest run of clearedbits is set to 1, above that is 0. So the high bit, tested by $\& 0x80$ can only be set when the entire byte is cleared.

For multiple bytes we can see that this works fine if we ignore borrow from one byte to another. Borrow can only happen if one byte started out as zero anyway though, in which case the lower byte will properly register as non-zero.

Benefit from hardware implementation:

Some architectures have both a population count instruction, and a find most significant bit AKA $\log_2$. If yours doesn't, these might help:

Population count

Naive:

```
unsigned int iterative_popcount(unsigned int b) {
    unsigned int n;
    for (n = 0; b != 0; n++, b &= (b - 1));
    return n;
}
```

works in $O(\text{number of set bits})$

```
#define m1 ((unsigned_64) 0x5555555555555555)
#define m2 ((unsigned_64) 0x3333333333333333)

unsigned int non_iterative_popcount(const unsigned_64 b) {
    unsigned_32 n;
    const unsigned_64 a = b - ((b >> 1) & m1);
    const unsigned_64 c = (a & m2) + ((a >> 2) & m2);
    n = ((unsigned_32) c) + ((unsigned_32) (c >> 32));
    n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);
    n = (n & 0xFFFF) + (n >> 16);
    n = (n & 0xFF) + (n >> 8);
    return n;
}
```

can be extended to any size, and is then $O(\log n)$, of course, for this case it is $O(1)$, since the number of bits are constant.

find MSB, or $\log_2$

naive. If you know more about your distribution, you can do stuff like check bytes (or words) at a time. For 8-bit bytes you should probably use a lookup table.

```
unsigned int iterative_log2(unsigned int b) {
    unsigned int n;
    for (n = 0; b != 0; n++, b >>= 1);
    return n;
}
```

from JLM:

```
#define BITCOUNT(x)      (((BX_(x)+(BX_(x)>>4)) & 0x0F0F0F0F) % 255)

#define BX_(x)             ((x) - (((x)>>1)&0x77777777)           \
                             - (((x)>>2)&0x33333333)           \
                             - (((x)>>3)&0x11111111))

unsigned int wordrev(unsigned int n) {
    n = ((n >> 1) & 0x55555555) | ((n << 1) & 0xaaaaaaaa);
    n = ((n >> 2) & 0x33333333) | ((n << 2) & 0xcccccccc);
    n = ((n >> 4) & 0x0f0f0f0f) | ((n << 4) & 0xf0f0f0f0);
    n = ((n >> 8) & 0x00ff00ff) | ((n << 8) & 0xff00ff00);
    n = ((n >> 16) & 0x0000ffff) | ((n << 16) & 0xffff0000);
    return n;
}
```